Project File System Report

Group 43

Name	Autograder Login	Email
Sarvagya Somvanshi	student176	sarvagya@berkeley.edu
Nathan Canera	student282	nathancanera@berkeley.edu
John Chang	student271	jchang04@berkeley.edu
Shengxiang Lin	student35	linshengxiang@berkeley.edu

Changes

Buffer Cache

We made significant changes to the buffer cache's synchronization. Originally, we would use the global cache lock to both view and edit metadata, and the local entry lock to view and edit data. However, we ran into issues where we would use the global lock to view metadata, and then try to read from a cache sector, only for the sector to be replaced with another sector. As a result, we changed the utilization of the local entry lock for editing metadata and viewing / editing data.

Extensible Files

As expected, we didn't need any changes for extensible files. Our design dock was very good and led us to a smooth and fun experience implementing extensible files.

Subdirectories

Subdirectories was quite the go around. First, I'll start with how our structs changed. In process.h, we removed the is_root attribute because we never needed it. Our inode structs changed a bunch because of buffer cache and extensible files implementations. From the subdirectories portion of the design doc though, we kept the `is_dir` boolean inside of `inode_disk` and used it often. Our `file_descriptor` struct remained as designed as well. In terms of implementing each of the syscalls and adjusting those that needed adjustment, we kept most of our relatively basic design the same. Instead of adjusting `sys_open` directly, we changed `filesys_open` to handle the path traversing and also open according to whether or not the file is a directory or not. We followed a similar process for remove and create. Our `sys_read` and `sys_write` stayed the same as well. For `sys_close` we kept the same design except we used `dir_close` instead of `dir_remove`. One thing that did change a decent bit was that when it came to determining whether or not a file path was relative or absolute, instead of trying both and seeing which one worked, we just checked if the first character of the path was a `/` or not and traversed with `get_next_part`. Our `chdir`, `mkdir`, `readdir`, `isdir`, and `inumber` stayed the same as in our design doc.

Reflection

Project 3 was a very productive experience for us as we gained a deep understanding of how file systems are built. We learned the importance of having a good preliminary design, as we spent considerable effort in backtracking our buffer cache and subdirectory implementations. However, we were able to make progress by parallelizing our efforts while working together to debug and close gaps in understanding.
Implementing the underlying inode functionality for extensible files (by Sarvagya) took considerable debugging from problems like threads self-destructing, but ultimately we overcame this hurdle. Merging the buffer cache (by John) and the extensible files portions of our code was easier than expected, as both parts were almost fully functional prior to merging. Implementing the directory syscalls (by Nathan) took longer as our approach had to be rethought to accommodate for changes that the extensible files made to the code. To overcome this challenge, we debugged the subdirectory code as a group. Finally, we wrote tests to inspect the efficiency and functionality of our buffer cache (by Shengxiang). Overall, our combined efforts and mutual support led us to complete this project right in the nick of time.

Testing

test-hit-rate

To test whether or not the cache is actually caching, we want to check how many reads we make to the disk in our initial read of the file, and also in our second read of the file. This test reads in a file of size 64KiB byte-by-byte, prints out the block statistics (read count and write count for the file system), closes the file, opens it again and the prints out the block statistics again. We expect the block statistics to NOT change when opening and closing, since all changes are cached and are not flushed to disk yet.

```
#include <syscall.h>
#include "tests/lib.h"
#include "tests/main.h"
#define FILE_SIZE 64 * 1024
static char buf_a[FILE_SIZE];
static char buf_b[FILE_SIZE];
static void write_bytes(const char* file_name, int fd, const c
har* buf, size_t* ofs) {
  if (*ofs < FILE_SIZE) {
    size_t ret_val;
    ret_val = write(fd, buf + *ofs, 4);
    *ofs += 4;
  }
}
static void read_bytes(const char* file_name, int fd, const ch
ar* buf, size_t* ofs) {
  if (*ofs < FILE_SIZE) {
    size_t ret_val;
    ret_val = read(fd, buf + *ofs, 4);
    *ofs += 4;
  }
}
void test_main() {
  int fd_a;
  size_t ofs_a = 0;
  random_init(0);
  random_bytes(buf_a, sizeof buf_a);
  CHECK(create("a", 0), "create \"a\"");
  CHECK((fd_a = open("a")) > 1, "open \"a\"");
  read_bytes("a", fd_a, buf_a, &ofs_a);
  msg("Checking block statistics after reading \"a\" ");
  stats();
  close(fd_a);
  CHECK((fd_a = open("a")) > 1, "re-open \"a\"");
  ofs_a = 0;
  read_bytes("a", fd_a, buf_a, &ofs_a);
  msg("Checking block statistics after re-reading \"a\" ");
  stats();
  msg("close \"a\"");
  close(fd_a);
  return;
}
```

output:

```
+ (test-hit-rate) begin
+ (test-hit-rate) create "a"
+ (test-hit-rate) open "a"
+ (test-hit-rate) Checking block statistics after reading "a"
+ hdb1 (filesys): 357 reads, 286 writes
+ hda2 (scratch): 297 reads, 2 writes
+ (test-hit-rate) re-open "a"
+ (test-hit-rate) Checking block statistics after re-reading "a"
+ hdb1 (filesys): 357 reads, 286 writes
+ hda2 (scratch): 297 reads, 2 writes
+ (test-hit-rate) close "a"
+ (test-hit-rate) end
```

If my kernel called block_read instead of cache_read, it would increase the number of reads to filesys after reopening “a” and reading from “a” again.
Similarly, if the kernel flushes cached blocks to disk prematurely when the file is closed and reopened, the system will treat the second read as an uncached access, leading to additional disk reads.

test-coalesce
We wrote this test to see if the cache would coalesce writes to the block sector. The test first writes a 64KiB file to memory and then reads. We want to see if the total number of writes is a multiple of 128 (since 64KiB = 128 blocks).

```
#include <syscall.h>
#include "tests/lib.h"
#include "tests/main.h"
#define FILE_SIZE 64 * 1024
static char buf_a[FILE_SIZE];
static char buf_b[FILE_SIZE];
static void write_bytes(const char* file_name, int fd, const char* buf, size_t* ofs) {
  if (*ofs < FILE_SIZE) {
    size_t ret_val;
    ret_val = write(fd, buf + *ofs, 4);
    *ofs += 4;
  }
}
static void read_bytes(const char* file_name, int fd, const char* buf, size_t* ofs) {
  if (*ofs < FILE_SIZE) {
    size_t ret_val;
    ret_val = read(fd, buf + *ofs, 4);
    *ofs += 4;
  }
}
void test_main() {
  int fd_a;
  size_t ofs_a = 0;
  random_init(0);
  random_bytes(buf_a, sizeof buf_a);
  CHECK(create("a", 0), "create \"a\"");
  CHECK((fd_a = open("a")) > 1, "open \"a\"");
  msg("Checking block statistics after opening \"a\" ");
  stats();
  msg("write \"a\"");
  while (ofs_a < FILE_SIZE) {
    write_bytes("a", fd_a, buf_a, &ofs_a);
  }
  msg("Checking block statistics after writing to \"a\" ");
  stats();
  while (ofs_a < FILE_SIZE) {
    read_bytes("a", fd_a, buf_a, &ofs_a);
  }

  msg("Checking block statistics after reading\"a\" ");
  stats();
  msg("close \"a\"");
  close(fd_a);
  return;
}
```

output:

```
+ (test-coalesce) begin
+ (test-coalesce) create "a"
+ (test-coalesce) open "a"
+ (test-coalesce) Checking block statistics after opening "a"
+ hdb1 (filesys): 357 reads, 286 writes
+ hda2 (scratch): 297 reads, 2 writes
+ (test-coalesce) write "a"
+ (test-coalesce) Checking block statistics after writing to
"a"
+ hdb1 (filesys): 493 reads, 385 writes
+ hda2 (scratch): 297 reads, 2 writes
+ (test-coalesce) Checking block statistics after reading"a"
+ hdb1 (filesys): 493 reads, 385 writes
+ hda2 (scratch): 297 reads, 2 writes
+ (test-coalesce) close "a"
+ (test-coalesce) end
```

If my kernel performed a separate disk write for each byte written instead of aggregating writes into blocks and writing them as a single operation, then the test case would output a much higher write count than the expected ~128 writes.
If the kernel prematurely evicts buffer cache blocks before they are fully written or aggregated (e.g., due to a poor eviction policy or failure to prioritize frequently used blocks), writes to the same block could result in multiple separate writes to the disk.

Writing tests for Pintos provided invaluable insights into the complexities of kernel development and the critical role of rigorous testing in ensuring system stability and performance. We used our tests to as a sanity check to verify if our cache was actually working before proceeding to integrating the cache with the rest of our code. This ensured that we had an easy integration experience instead of suffering through combined buffer cache and subdirectory bugs.